На этом занятии познакомимся с константными методами класса и посмотрим, как они определяются и для чего они нужны.

Пусть у нас имеется следующий класс:

```language-cpp
class Point {
    int x {0}, y{0};

public:
    Point(int a = 0, int b = 0) : x(a), y(b)
        { }

    void set_coords(const Point& p)
    {
        x = p.x; y = p.y;
    }

    void get_coords(int& x, int& y)
        { x = this->x; y = this->y; }
};
```

Использовать его можно следующим образом:

```language-cpp
int main()
{
    Point pt, pt2(1, 2);

    pt.set_coords(pt2);

    int x, y;
    pt.get_coords(x, y);
    std::cout << x << " " << y << std::endl;

    return 0;
}
```

Здесь метод set_coords принимает константную ссылку на объект класса Point и задает значения координат текущего объекта. Так как параметр метода имеет модификатор const, то это сигнализирует программисту, что переданный объект никак меняться в методе set_coords не будет. И, действительно, компилятор отслеживает все эти моменты. Например, если попытаться занести в переменные x, y переданного объекта какие-либо значения:

```language-cpp
    void set_coords(const Point& p)
    {
        x = p.x; y = p.y;
        p.x = 1;
        p.y = 2;
    }
```

то возникнет ошибка на этапе компиляции программы. Из-за того, что параметр p является константным, значение его полей можно только читать, но не менять. **Однако, в отличие от обычных примитивных типов данных, объекты класса можно менять и через вызовы его методов**. Например, так:

```language-cpp
    void set_coords(const Point& p)
    {
        x = p.x; y = p.y;
        p.set_coords(Point(10, 20));
    }
```

И здесь компилятор предусмотрительно запретит такой вызов из-за константного объекта. Но он также запретит и вполне безобидный вызов метода для получения координат:

```cpp
void set_coords(const Point& p)
{
x = p.x; y = p.y;
p.get_coords(x, y);
}
```

```
Мы знаем, что метод get_coords не меняет состояния объекта, а значит, вполне мог бы быть вызван через константный объект. Как сообщить компилятору, что этот или какой-либо другой метод можно вызывать для константного объекта? Для этого **Бьерн Стауструп предложил такие методы помечать особым образом: записывать после их объявления ключевое слово const следующим образом**:

 ```cpp
   void get_coords(int& x, int& y) const
        { x = this->x; y = this->y; }
```

Такие методы получили название **константных** и могут вызываться константными объектами класса. Обратите внимание, что ключевое слово
```

const записано после прототипа функции, но до ее тела. Именно так следует определять константные методы. Теперь программа успешно компилируется с вызовом метода get_coords.

Также не следует путать запись ключевого слова const в начале метода, с таким же словом const в конце прототипа, например:

```cpp
const int get_x() const
{ return x; }
```

Первое слово const относится к возвращаемому типу int и означает, что метод get_x возвращает координату в виде целочисленной константы. А второе слово const, как раз определяет константный метод, который можно впоследствии вызывать через константные объекты класса Point. Эти два определения нужно четко различать.

Вообще в практике программирования рекомендуется все методы, которые не меняют состояние объектов класса, помечать, как константные. Тогда программист, пользователь класса, сможет в полной мере, свободно и корректно использовать его в своем проекте.

Также, **обратите внимание, что константные методы могут, в свою очередь, вызывать только другие константные методы**. Например, следующий код будет корректен:

```cpp
class Point {
    int x {0}, y{0};

public:
    Point(int a = 0, int b = 0) : x(a), y(b)
        { }

    void set_coords(const Point& p)
    {
        p.get_coords(x, y);
    }

    const int get_x() const
        { return x; }
```

```
    const int get_y() const
        { return y; }

    void get_coords(int& x, int& y) const
        { x = get_x(); y = get_y(); }
};
```

Но, если хотя бы один из методов get_x или get_y не будет константным, то компилятор сообщит об этом ошибкой на этапе компиляции программы.

Следующая особенность константных методов. **Если они возвращают указатель или ссылку на переменную, то нужно такие типы предварять ключевым словом const**:

```language-cpp
    const int& get_x() const
        { return x; }

    const int* get_y() const
        { return &y; }

    void get_coords(int& x, int& y) const
        { x = get_x(); y = *get_y(); }
```

И понятно, почему. Если бы была возвращена обычная ссылка или указатель:

```cpp
int& get_x() const
{ return x; }


int* get_y() const
{ return &y; }
```

то через них можно было бы менять значение полей объекта, что недопустимо при вызове константных методов. Но, если возвращается копия объекта:

```cpp
int get_x() const
        { return x; }
```

```
    int get_y() const
        { return y; }
```

то ключевое слово const можно не использовать, т.к. через копию данных нельзя изменить состояние объекта.

Ключевое слово mutable

В очень редких случаях, которые следует избегать на практике, некоторые переменные объекта можно помечать специальным ключевым словом mutable, позволяющее изменять переменную константного объекта. Это делается следующим образом:

```language-cpp
class Point {
    int x {0}, y{0};

public:
    mutable int count_call {0}; // число вызовов различных методов текущего объекта

    Point(int a = 0, int b = 0) : x(a), y(b)
        { }

    void set_coords(const Point& p)
    {
        count_call++;
        p.get_coords(x, y);
    }

    int get_x() const
        { count_call++; return x; }

    int get_y() const
        { count_call++; return y; }

    void get_coords(int& x, int& y) const
        { count_call++; x = get_x(); y = get_y(); }
};
```

Здесь переменная count_call помечена ключевым словом mutable и, соответственно, может совершенно спокойно изменяться во всех методах класса Point, включая константные. Если убрать слово mutable, то компилятор выдаст ошибку об изменении переменной в константных методах.

Однако, как я уже говорил, это не рекомендуемая практика, которую следует избегать. **Использование подобных переменных нарушает принцип неизменности состояния константного объекта и может легко привести к непредвиденным последствиям**.